electronics

MDPI

Article

RegFuzz: A Linear Regression-Based Approach for Seed Scheduling in Directed Fuzzing

Mingmin Lin, Yingpei Zeng * and Yang Li

School of Cyberspace, Hangzhou Dianzi University, Hangzhou 310000, China; lmmwxy@hdu.edu.cn (M.L.); chrisly@hdu.edu.cn (Y.L.)
* Correspondence: yzeng@hdu.edu.cn

Abstract: Directed fuzzing aims to focus on fuzzing specific locations within a target program to enhance the efficiency of vulnerability discovery. However, directed fuzzing may yield fewer vulnerabilities and obtain lower code coverage when the specified locations have little to no vulnerabilities. Additionally, the existing directed fuzzing approaches often overlook the differences in variable values when calculating distances between seeds and specific locations. In order to address these issues, this paper introduces RegFuzz, a method that improves seed scheduling in directed fuzzing. RegFuzz utilizes a linear regression model to predict the effectiveness of a seed and allocates more fuzzing opportunities to efficient seeds. Specifically, first, RegFuzz defines several labels with the corresponding trainable weights for each seed. These labels encompass seed coverage, crash efficiency, seed distance, and more. In the calculation of seed distance, RegFuzz takes into account not only the basic block distance but also the *variable distance* contained within those basic blocks. Second, the linear regression model continually optimizes the label weights during fuzzing, and these optimized weights are employed to predict the effectiveness of seeds. In comparison with AFLGo, AFL, and AFL++, RegFuzz demonstrates higher code coverage and a more efficient bug-finding capability across seven real-world open-source programs.

Keywords: fuzz testing; directed fuzzing; linear regression; seed scheduling

Citation: Lin, M.; Zeng, Y.; Li, Y. RegFuzz: A Linear Regression-Based Approach for Seed Scheduling in Directed Fuzzing. *Electronics* **2023**, *12*, 3650. https://doi.org/10.3390/electronics12173650

Academic Editor: Manuel Mazzara

Received: 26 July 2023
Revised: 25 August 2023
Accepted: 28 August 2023
Published: 29 August 2023

1. Introduction

Directed fuzzing [1] is an advanced technique used in the field of fuzzing. Unlike general fuzzing [2–4], which aims to cover as much code and discover as many vulnerabilities as possible throughout the *entire* program, directed fuzzing [1,5] focuses on specific *parts* of the program to achieve code coverage and vulnerability discovery. Directed fuzzing is particularly suitable for scenarios such as verifying the historical patches of the PUT. This involves conducting intensive fuzzing on the patched code to validate the effectiveness of the patch and ensure that no new vulnerabilities are introduced [6–8]. In directed fuzzing, the CFG (control flow graph) and CG (call graph) of the program under test (PUT) are commonly utilized to calculate the distance between the basic blocks of the current seed and the target basic blocks. The technique emphasizes fuzzing seeds that show promise, often indicated by a shorter distance [1,5].

However, directed fuzzing does have certain limitations. First, if the specified part of the program code that users are targeting for vulnerability discovery contains little to no vulnerabilities, the majority of the fuzzing efforts dedicated to that part would be rendered ineffective, whereas the other areas of the program remain unexplored. As a result, the overall code coverage achieved may be low. In such situations, users may prefer directed fuzzers that automatically extend their scope of fuzzing to include other parts of the program as well. Second, most existing directed fuzzers, such as AFLGo [1], primarily consider superficial coverage information when scheduling seeds. For instance, directed fuzzers [1,9–11] employ various calculation methods to determine the distance between

the current basic blocks and the target basic blocks. However, they often overlook critical variable information within the target basic blocks. This variable information typically plays a pivotal role in triggering program vulnerabilities, implying that a vulnerable path not only needs to be covered but also requires specific values to be assigned to the variables [12].

In this paper, we present RegFuzz, a solution for seed scheduling in directed fuzzing [13]. RegFuzz effectively addresses the issue of low coverage in directed fuzzing and enhances vulnerability discovery efficiency through its dynamic linear regression-based prediction model. First, RegFuzz introduces four fuzzing labels for seeds: coverage efficiency, crash efficiency, seed distance (including *variable distance*), and fuzzing speed. Second, during the fuzzing process, each seed is assigned a score based on its label values and the corresponding label weight values. Seeds with higher scores are given more opportunities for fuzzing. Finally, RegFuzz dynamically adjusts the label weights based on the fuzzing results, enabling the continuous evaluation of seed scores. In order to evaluate the performance of RegFuzz, we conduct a comparative analysis using state-of-the-art fuzzers, including AFL [2], AFLGo [1], and AFL++ [14], using seven open-source programs. The results demonstrate that RegFuzz achieves superior performance in terms of code coverage and vulnerability discovery. In summary, this paper makes the following contributions:

- We have devised four distinct labels for each seed, capturing essential aspects such as code coverage, seed distance, execution efficiency, and vulnerability discovery efficiency.
- We have introduced a linear regression model to train and assign labels to seeds, thereby granting more fuzzing opportunities to those seeds with promising potential.
- We have implemented a fully functional prototype of RegFuzz, applied it to fuzz seven open-source programs, and conducted a comparative analysis against three state-of-the-art fuzzers. The results demonstrate the superior performance of RegFuzz when compared to other existing fuzzers.

2. Background

2.1. American Fuzzy Lop (AFL)

AFL [2], developed by security researcher Michal Zalewski, is a coverage-guided fuzzing system renowned for its effectiveness. It employs a coverage-based approach to guide seed mutation, thus enhancing code coverage and increasing the likelihood of vulnerability detection. By leveraging program coverage feedback, AFL effectively explores the program's logic execution space. This concept of greybox fuzzing has inspired researchers in the security industry, leading to the development of similar efficient greybox fuzzers like LibFuzzer [15], VUzzer [12], and BFF [16]. The basic framework of AFL is illustrated in Figure 1. The fuzzer comprises several key modules, including the program monitor module, seed generator module, crash detector module, and crash filter module. Each of these modules serves a specific role and is explained as follows:

- **Program monitor**: This module provides an execution environment for the target programs and monitors their runtime information during the fuzzing process. It captures the program's running status and, when a seed triggers a program crash, preserves the seed for vulnerability analysis.
- **Seed generator**: The seed generator serves as the core module of the fuzzing system and significantly influences the fuzzing outcomes. It relies heavily on seed scheduling and mutation strategies, which involve selecting promising seeds and determining how to mutate them effectively.
- **Crash detector and filter**: The crash detector and filter play a critical role in the fuzzing process. Treating all abnormal program states as vulnerabilities would result in a high false positive rate. In order to address this, the fuzzing system preserves the seed that triggers an exception for subsequent replay verification, ensuring a low false alarm rate. The operation of vulnerability detectors may vary across different operating

system platforms. For instance, on Linux, the fuzzer detects and filters vulnerabilities based on the specific signals triggered during program crashes (such as SIGSEGV).

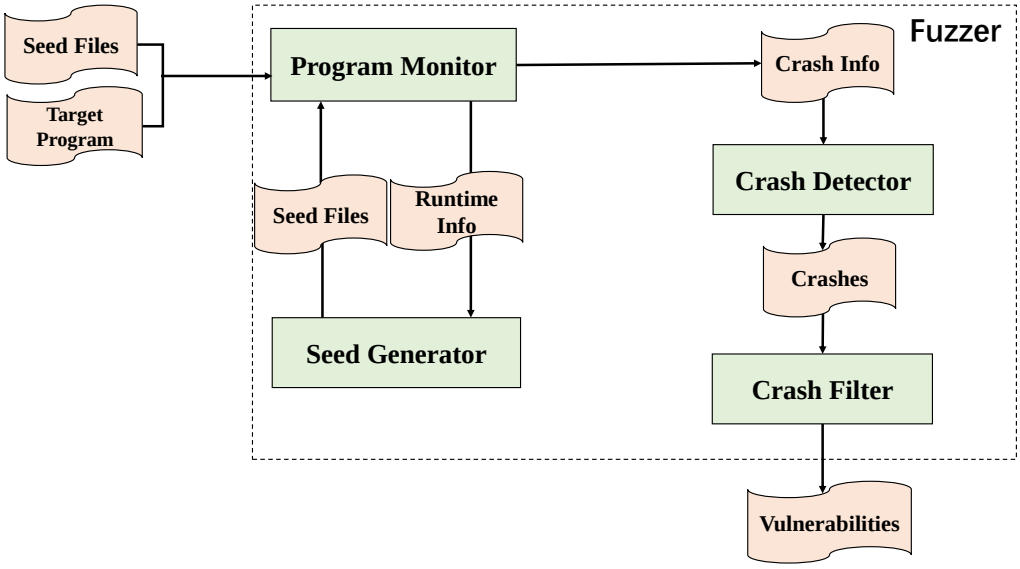


Figure 1. The framework of AFL.

2.2. Linear Regression Model

Regression analysis is a modeling technique that allows us to study the relationship between a target variable and predictor variables. Linear regression is one of the widely used regression models. It assumes a linear correlation between the target value and the predictor labels, satisfying a multivariate linear equation. By constructing a loss function, the goal of linear regression is to determine the optimal parameters that minimize the loss function. The regression function can be expressed using Equation (1):

$$\widehat{y} = h_\theta(x) = \theta \cdot x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n, \tag{1}$$

where $\widehat{y}$ represents the predicted value, θ represents the label weight vector, x represents the label value vector, and $\theta(x)$ represents the loss function. In this equation, the independent variable x is assumed to be known. The objective of linear regression is to train a set of label weight vectors θ that best align with the observed data, enabling the linear regression model to make more accurate predictions. In order to accomplish this, we need to determine the parameter values of the θ vector in the linear model based on the available data points.

Each training model is assessed using an evaluation criterion, and to evaluate the training outcomes of linear regression models, the minimum mean square error (MSE) equation is commonly employed. The MSE equation is defined as Equation (2):

$$MSE = (X, h_\theta) = \frac{1}{m} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)}), \tag{2}$$

where m represents the number of instances, θ^T represents the transposition of the label weight vector of each instance, $x^{(i)}$ represents the label vector of the ith instance, and $y^{(i)}$ represents the actual value of each instance.

After completing the model training, the model will be updated based on the label weight values corresponding to the set of training models with the lowest MSE, as shown in Equation (3):

$$\theta = (X^T X)^{-1} X^T Y, \tag{3}$$

where X represents the combination of each instance in training, X^T represents the transposition of X, and Y represents the actual value.

During the fuzzing process, the effectiveness of the seeds is typically assessed based on multiple labels, such as code coverage, seed distance, and more. These labels impact the fuzzing process from different perspectives, aligning well with the influence of multiple independent variables on the regression outcomes. Furthermore, as numerous mutations are applied to all seeds during fuzzing, it creates a natural training environment that is highly suitable for training the regression model.

3. Motivation

On the one hand, as mentioned earlier, directed fuzzing may not automatically switch to fuzzing other unspecified parts of the program swiftly, even when the specified part of the program contains few or no vulnerabilities. This limitation can result in low code coverage and a reduced number of discovered vulnerabilities. Directed fuzzing locks the locations in advance during the compilation phase and conducts in-depth testing near the targeted basic block locations during the fuzzing execution. For instance, seeds with smaller basic block distances are given higher priority and allocated more resources for fuzzing [1]. While this approach works well when the specified locations contain numerous vulnerabilities, it becomes less effective when the specified locations have fewer or no vulnerabilities.

On the other hand, most existing directed fuzzers solely focus on coverage information when selecting seeds for fuzzing. However, in many cases, the values of variables play a crucial role in triggering vulnerabilities. Let us consider the code snippet of CVE-2023-28856 shown in Listing 1 (present in Redis before v7.0.11) as an example. This vulnerability is only triggered when the variable `incr` has a large or NaN (not-a-number) value. Otherwise, all lines in the function execute without any issues. Thus, if only code coverage is considered while ignoring variable values, it becomes challenging to discover such vulnerabilities.

Listing 1. Simplified code snippet of CVE-2023-28856.

```
1  void hincrbyfloatCommand(client *c) {
2    //...
3    if (getLongDoubleFromObjectOrReply(c,c->argv[3],&incr,NULL) != C_OK) return;
4
5    // Later added code for fixing the vulnerability
6    //if (isnan(incr) || isinf(incr)) {
7    //  addReplyError(c,"value is NaN or Infinity");
8    //  return;
9    //}
10
11   if ((o = hashTypeLookupWriteOrCreate(c,c->argv[1])) == NULL) return;
12   //...
13 }
```

4. The RegFuzz Approach

4.1. Overview

Essentially, RegFuzz leverages linear regression to predict the efficiency of seeds. It introduces four fuzzing labels for seeds: coverage efficiency, crash efficiency, seed distance, and fuzzing speed. Although seed distance aligns with existing directed fuzzing techniques, we also incorporate the consideration of variable distance. Additionally, we believe the other three labels are crucial for the success of fuzzing, and additional labels can be included if required. During the fuzzing process, each seed is assigned a score based on the product of its label values and the corresponding label weight values (Equation (1)). Seeds with higher scores receive more opportunities for fuzzing. Furthermore, RegFuzz dynamically adjusts the label weights based on the fuzzing results, allowing for the dynamic evaluation of seed scores. As a result, RegFuzz explores other parts of the program when directed fuzzing encounters limitations, thus improving the overall efficiency of vulnerability discovery.

The framework of RegFuzz is illustrated in Figure 2. Prior to the fuzzing execution, RegFuzz conducts program analysis to obtain the call graph (CG) and control flow graph (CFG). These CGs and CFGs are utilized to calculate both the basic block distance and variable distance within the program. During the instrumentation phase, the fuzzer incor-

porates the distance information into the target program. At runtime, RegFuzz records the basic blocks traversed by each seed and monitors the current values of the variables. It uses the instrumented distance information to calculate the seed distance while also gathering the necessary information to compute the values for the remaining three seed labels. Within the fuzzing loop, RegFuzz employs regression predictions to generate seed scores, enabling the scheduling of seeds based on their scores.

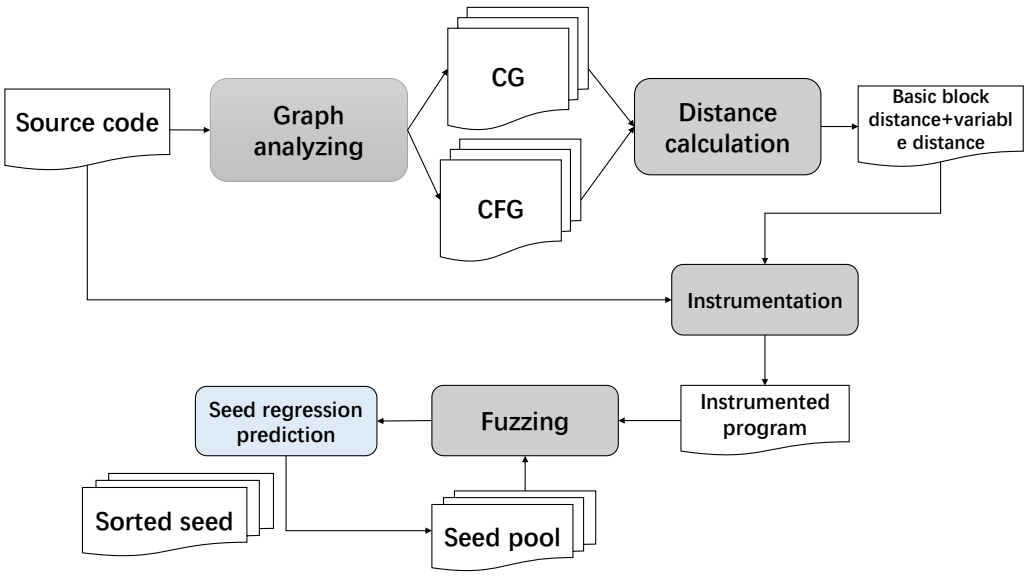


Figure 2. The framework of RegFuzz.

4.2. Seed Labels

According to the overall introduction of the RegFuzz framework in the previous subsection, RegFuzz mainly implements a seed scheduling strategy in directed fuzzing, and the most important basis is the definition of seed labels. Their details are shown in Table 1.

Table 1. Seed labels and their definitions.

Seed Labels	Description
Seed coverage efficiency	Coverage generated by seed during fuzzing
Seed distance	Distance between seed and target basic blocks
Seed crash efficiency	Crashes generated by seed during fuzzing
Seed execution speed	Average speed of seed fuzzing execution

The calculation method of seed coverage is shown in Equation (4), where N_s is the number of new seeds generated by the seed, and F_t is the number of fuzzing times of the seed.

$$Cov_e = \frac{N_s}{F_t} \tag{4}$$

The calculation method of seed distance is shown in Equation (5), which equals the sum of the basic block distance B_d from the seed to target basic blocks and the variable distance V_d.

$$D = B_d + V_d \tag{5}$$

The calculation method of seed crash efficiency is shown in Equation (6), where C_n is the number of crashes found.

$$C_e = \frac{C_n}{F_t} \tag{6}$$

The execution speed of a seed is calculated as shown in Equation (7), where T_f is the time that the seed consumed in fuzzing.

$$F_s = \frac{F_t}{T_f} \tag{7}$$

Among these labels, the seed coverage efficiency, crash efficiency, and execution speed can be directly calculated and obtained in the fuzzing. When calculating the seed distance, it is necessary to know in advance the distance between all basic blocks and the target basic blocks, as well as the distance between all monitored variables and their thresholds. The concepts of basic block distance and variable distance and their specific calculation methods will be described in detail in Section 4.2.1.

4.2.1. Basic Block Distance and Variable Distance

In the construction of seed labels, the most complicated one is the seed distance computation. RegFuzz incorporates AFLGo's method for calculating seed basic block distance [1] and devises a customized calculation approach for variable distance. By evaluating both the basic block distance and variable distance, RegFuzz can explore deeper into the program, providing a more accurate assessment of how closely a seed is positioned to the target basic blocks. It is important to note that the calculation of seed distance requires prior knowledge of the CG and CFG of the program. Before the fuzzing process starts, LLVM is used to traverse the CG and CFG, capturing information on all basic blocks and functions. This information is then leveraged for subsequent seed distance calculations. The detailed calculation methods for these two types of distances are as follows:

The calculation of the seed basic block distance needs to use the function level distance. The function level distance is calculated as Equation (8) [1]:

$$d_f(n, T_f) = \left[\sum_{t_f \varepsilon R(n, T_f)} d_f(n, t_f)^{-1} \right]^{-1}, \tag{8}$$

where T_f represents a set of target functions, $d_f(n, T_f)$ represents the harmonic average of the sum of distances from function n to any reachable target function, and $R(n, T_f)$ represents the set of functions from n to the target function T_f in the CG. The distance between function n and target function set T_f is calculated according to a path in the CG graph, which is the shortest edge in the CG from n to t_f. The harmonic average of n to the sum of all t_f distances is then calculated.

Then, for each basic block in the function, the basic block level distance can be calculated using Equation (9). The calculation of basic block level distance is similar to the function level distance, and it is also based on the function level distance. For each basic block, we need to find the shortest path from the basic block to the target basic blocks in the CFG. Then, according to the position of the current basic block and target basic blocks in the CFG, there are three calculation methods [1]:

$$d_b(m, T_b) = \begin{cases} 0, \\ c \cdot min_{n \epsilon N(m)}(d_f(n, T_f)), \\ \left[\sum_{t \varepsilon T} (d_b(m, t) + d_b(t, T_b))^{-1} \right]^{-1}, \end{cases} \tag{9}$$

where $d_b(m, T_b)$ is the harmonic average of the total distance from m to the reachable target basic blocks in the CFG. T_b refers to the target basic block set, $N(m)$ is the set of all functions called by the current basic block m, and T is a set of basic blocks included in the CFG. From Equation (9), it can be seen that if the current basic block m is in the target basic blocks, then the basic block distance is 0. If the current basic block m is not in the target basic blocks but is within the current CFG range, the calculation rule is shown in the middle. In other cases (that is, the current basic block and the target basic blocks are in different CFGs), the basic

block distance is equal to the basic block distance of the two CFGs, adding the minimum distance to the target basic blocks.

When calculating variable distance, the calculation method is similar to the above calculation method. First, we need to find out the marked variable set V (including the function parameter variables and local variables contained in T_f) in T_b and record the threshold value corresponding to each variable. In the dynamic execution process, the fuzzer discovers the current value of a variable, divides it by its corresponding threshold value as the variable current distance, and sums up all variable distance as total variable distance V_d:

$$V_d = \sum_{t \epsilon V} \frac{t_{lim}}{t}, \tag{10}$$

where V is the marked variable set, t is the variable in the subset V, and t_{lim} is the threshold value of the current variable.

After implementing the above basic distance calculation method and dynamically instrumenting the program, the program will calculate the distance from the seed to the target basic blocks as Equation (11), combined with Equation (5):

$$D = B_d + V_d = d(s, T_b) + V_d = \frac{\sum_{m \epsilon \xi(s)} d_b(m, T_b)}{|\xi(s)|} + V_d, \tag{11}$$

where s is the current seed, $d(s, T_b)$ is the distance from the current seed to target basic blocks set T_b, $\xi(s)$ is the execution path of the seed (a basic block execution path of seed s), and V_d is variable distance.

After getting the distances of all seeds, in order to ensure fairness, the distances of all seeds will be normalized to values between [0,1], using the maximum seed distance max_D and the minimum seed distance min_D, like AFLGo.

4.3. The Main Fuzzing Loop of RegFuzz

The main fuzzing loop of RegFuzz is depicted in Algorithm 1. Note that RegFuzz trains and uses a seed prediction model at the same time during the fuzzing. The algorithm takes three inputs: an initial seed set S, an initial label matrix X containing one vector (one line) for each seed, and the initial label weight vector θ of the regression model. RegFuzz first calculates the score for each seed in the seed set S, according to the linear regression model (lines 1–3). This involves multiplying the four label values (Section 4.2) of each seed with the corresponding label weight values, as described in Equation (1). After that, it selects a seed, s, from the seed set and assigns energy according to the computed score of s (lines 5–6). Then, it starts to fuzz the seed, s, by generating inputs from the seed through mutations (lines 7–14). During fuzzing, if a mutated input, s', triggers a new crash, RegFuzz adds s' to the crash set and updates the label vector of the seed, s (lines 9–10). If the mutated input, s', has new code coverage, RegFuzz adds s' to the seed set as a new seed, updates the label vector of seed s, and adds the initial label vector of the new seed, s', to X (lines 11–13). After all seeds complete a fuzzing cycle, RegFuzz sorts all seeds according to their predicted scores (line 16), and the top 10% of seeds have a separate cycle of fuzzing (lines 18–20). Finally, RegFuzz dynamically optimizes the label weight vector, θ, when it has finished a fuzzing cycle, according to the MSE equation (Equation (3)), and goes back to update the scores of all seeds and start the next cycle of fuzzing (lines 21).

Algorithm 1 The main fuzzing loop of RegFuzz

Input: S: seed set, X: initial label matrix (one line for each seed), θ: initial label weight vector
Output: C: crash set

1: **for** each seed s in seed set S **do**
2: $s.score = \text{CALCULATEREGRESSION}(X_s, \theta)$ ▷ Seed score prediction
3: **end for**
4: **repeat**
5: $s = \text{SELECTSEED}(S)$
6: $\rho = \text{ASSIGNENERGY}(s)$
7: **for** $i = 1; i <= \rho; i++$ **do**
8: $s' = \text{MUTATION}(s)$
9: **if** the PUT crashes with s' **then**
10: put s' into C, $\text{UPDATE}(X_s)$
11: **else if** the PUT has new coverage with s' **then**
12: put s' into S, $\text{UPDATE}(X_s)$, $\text{ADD}(X, s')$
13: **end if**
14: **end for**
15: **until** one cycle finishes
16: $S = \text{SORT}(S)$
17: $S' = \text{TOP}(S)$
18: **for** each seed s in seed set S' **do**
19: execute lines 6 to 14
20: **end for**
21: $\text{UPDATEWEIGHT}(\theta)$, goto line 1 ▷ Update the prediction model

5. Evaluation

We implemented a prototype of RegFuzz based on the greybox fuzzer AFL [2] and reused some code from AFLGo [1] for seed distance calculations.

5.1. Experiment Configuration

Target programs: We tested RegFuzz on seven Linux open-source programs. These programs are all from AFLGo's experimental programs, and most of them also appear as experimental programs in other AFL-type fuzzers. The detailed information is shown in Table 2. All experiments were conducted without adding dictionaries.

Table 2. Target programs and fuzzing commands.

Target	Format	Command
cxxfilt	txt	./cxxfilt
giflib	gif	./gifsponge
libxml2	xml	./xmllint –valid –recover @@
objdump	elf	./objdump -SD @@
mjs	file	./mjs-bin -f @@
libming	swf	./swftophp @@
lrzip	lrz	./lrzip -t @@

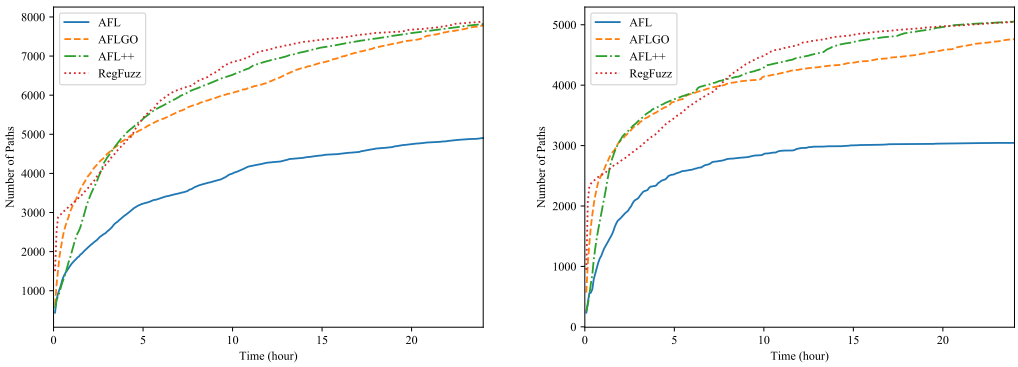
Fuzzers: We compared the performance of RegFuzz to three AFL-type fuzzers: AFL (v2.52b) [2], AFLGo [1], and AFL++ (v4.00c) [14]. The second one is a directed fuzzer, and the other two are state-of-the-art general fuzzers.

Experimental platform: All experiments were completed on a 64-bit machine with 16 cores (Intel (R) Core (TM) i7-10870H CPU @ 2.20 GHz) and 16 GB RAM. The operating system is Ubuntu 20.04. All target programs were fuzzed for 24 h, and each experiment was repeated five times.

Evaluation metrics: Our experiment uses relevant indicators to evaluate the performance of fuzzers, namely, unique path, code line coverage, and generated crashes. In addition, we also compare the average speed of vulnerability discovery. The experiment uses afl-cov [17] to measure the code line coverage of target programs and uses AddressSanitizer (ASAN) [18] to calculate the number of unique crashes.

5.1.1. Code Coverage Analysis

The path coverage results and code line coverage results of each fuzzer on different programs are shown in Table 3 and Figure 3. We can see that RegFuzz does not have the limitation of directed fuzzing since its performance is much better than the directed fuzzer AFLGo. Overall, RegFuzz is nearly consistent with AFL++ in path coverage and code line coverage. In terms of path coverage, RegFuzz is nearly 40% higher than its base fuzzer AFL and 15% higher than AFLGo. AFL++ combines the advantages of other excellent fuzzers (MOPT, AFLFast, etc.). It also has a more complicated seed mutation strategy and higher running speed. So, it performs very well in coverage. In almost all the programs that were tested, AFL++ is in first or second place. RegFuzz's seed prediction model plays an important role in directed fuzzing, and its path coverage is much higher than AFLGo's. In all seven program evaluations, the overall average path coverage of RegFuzz is 15% higher than that of AFLGo, and the average code line coverage is about 2% higher. This is because AFLGo is a directed fuzzing fuzzer. It does too many invalid mutations and tests near the target basic blocks. However, when RegFuzz performs directed fuzzing on target code blocks but does not catch any crashes or new coverage, the label weight of the seeds over a short distance is reduced, and according to the seed prediction model, seeds will be tested at a large distance from the target code, which improves the performance of directed fuzzing. In terms of code line coverage, RegFuzz has the highest code line coverage in `cxxfilt`, `libxml2`, and `lrzip`, and AFL++ has the highest code line coverage in the other four programs. Although AFL++ shows the best line coverage in a higher number of programs, it only has a small lead over RegFuzz. Therefore, based on the coverage experiment results, RegFuzz has good coverage performance.



(**a**) cxxfilt (**b**) giflib

Figure 3. *Cont.*

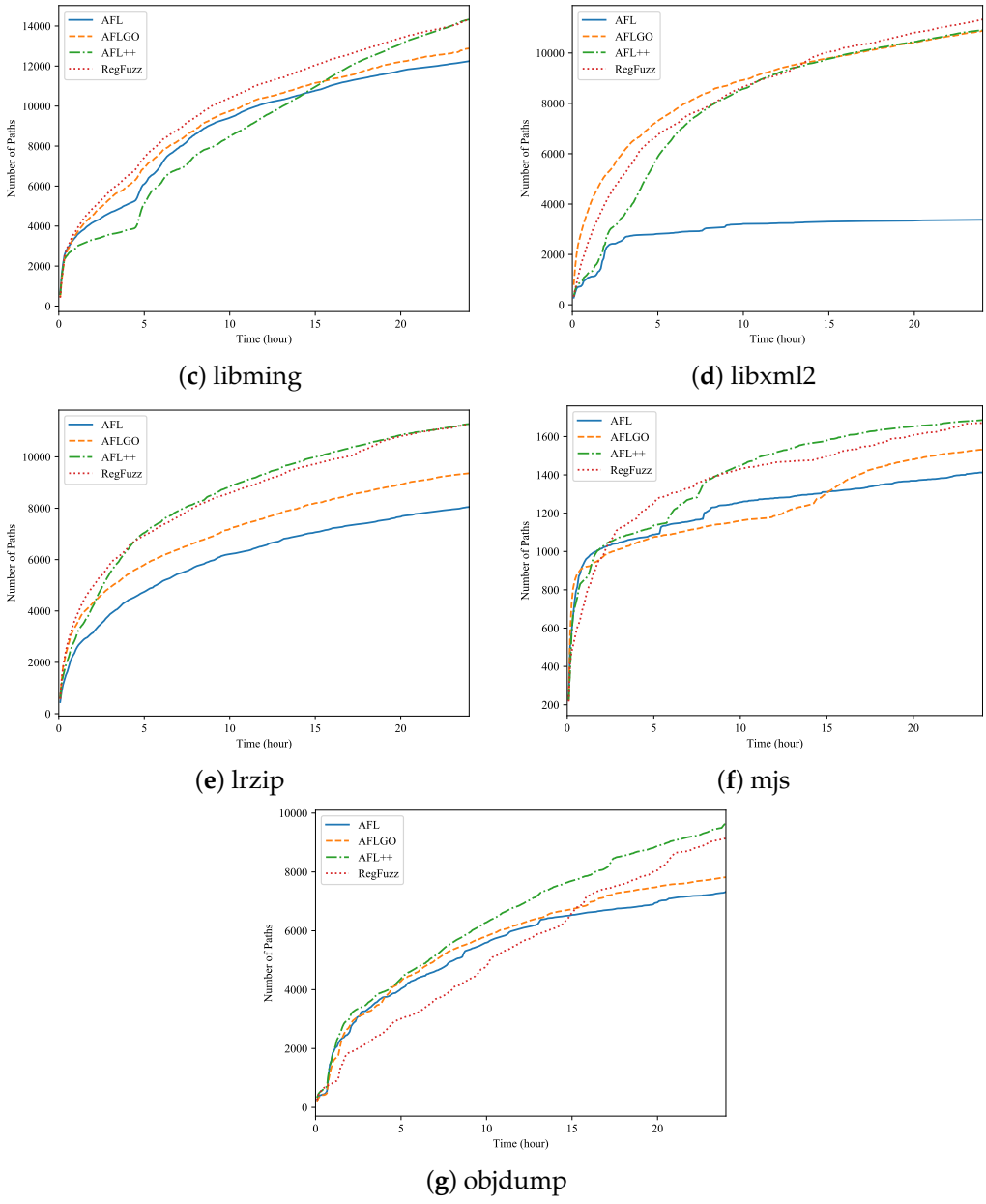


Figure 3. Average paths that AFL, AFL++, AFLGo, and RegFuzz found over five runs in 24 h.

5.1.2. Vulnerabilities Analysis

The vulnerability discovery of all fuzzers is shown in Table 4. RegFuzz is better than AFL and AFL++ and is slightly better than AFLGo in terms of vulnerability detection. This is because directed fuzzers make it easier to generate crash inputs in the target code. The result clearly shows that RegFuzz retains the benefits of directed fuzzing. In all programs that detect vulnerabilities, such as `cxxfilt`, `objdump`, and `lrzip`, RegFuzz's vulnerability detection capability is stable and higher than AFL and AFL++, which reflects the advantage of directed fuzzing. In `objdump`, the number of crashes detected by RegFuzz and the number of crashes after deduplication are higher than that of AFLGo, which reflects the fact that RegFuzz's seed scheduling plays an important role and enables RegFuzz to find vulnerabilities not only in the target code but also outside of the target code. For `libxml2` and `mjs`, none of the fuzzers in the experiment found any crashes because we used the latest software versions of the two pieces of software, and 24 h short-term fuzzing is not enough time to find any crashes. The results of the vulnerability experiment show that RegFuzz still has good ability in terms of vulnerability detection.

In order to better evaluate RegFuzz's vulnerability detection capabilities, Table 5 shows the detection efficiency of the same vulnerabilities for different programs. All fuzzers only detected the same vulnerability in four programs. These vulnerabilities have their corresponding CVE (common vulnerabilities and exposure) numbers, which are CVE-2016-4487 (`cxxfilt`), CVE-2018-8807 (`libming`), CVE-2017-8392 (`objdump`), and CVE-2018-11496 (`lrzip`), respectively. Table 5 shows that the CVE vulnerability detection efficiency (i.e., PoC generation) of AFLGo and RegFuzz is much higher than that of AFL and AFL++, which further confirms that RegFuzz retains the vulnerability discovery ability of directed fuzzing in specified locations. For example, in `cxxfilt`, `libming`, and `objdump`, AFL and AFL++ detect related vulnerabilities at the same time, but AFL++ is slightly earlier than AFL. In contrast, AFLGo and RegFuzz both complete the generation of PoC seeds within one hour. During fuzzing, they can quickly generate relevant seeds by calculating seed distance to traverse these basic blocks and trigger vulnerabilities to generate corresponding PoC seeds. For `lrzip`, RegFuzz detected CVE-2018-11496 in 1 h and 38 min, while AFLGo detected the vulnerability in 2 h and 10 min. This proves that the monitoring of critical variables in target basic blocks plays a positive role.

Therefore, RegFuzz has a relatively stable performance boost, both in terms of the number of vulnerabilities and the speed of vulnerability detection.

Table 3. The average of the evaluation metrics of the seven target programs.

Targets	Paths				Line Coverage			
	AFL	**AFL++**	**AFLGo**	**RegFuzz**	**AFL**	**AFL++**	**AFLGo**	**RegFuzz**
cxxfilt	4903	7818	7778/0	**7881**	16.92%	21.01%	20.53%	**21.18%**
giflib	3044	**5052**	4762	5048	13.16%	**17.27%**	15.03%	17.13%
libxml2	3375	10,910	10,883	**11,330**	42.28%	73.12%	72.25%	**73.36%**
objdump	7323	**9626**	7822	9147	40.07%	**41.93**%	40.31%	41.46%
libming	12,244	**14,328**	12,881	14,280	33.67%	**37.05%**	35.12%	37.05%
mjs	1412	**1686**	1533	1671	7.21%	**7.62%**	7.21%	7.56%
lrzip	8050	**11,280**	9362	11,266	44.56%	48.88%	45.23%	**48.88%**

Table 4. The number of crashes found by the fuzzers and the number of vulnerabilities after deduplication by ASAN.

Target	AFL	AFL++	AFLGo	RegFuzz
cxxfilt	36/2	47/2	325/4	244/4
giflib	2/1	14/2	82/3	80/4
libxml2	0	0	0	0
objdump	15/2	23/2	88/2	106/3
libming	1/1	1/1	16/3	8/2
mjs	0	0	0	0
lrzip	0	7/1	34/2	40/2

Table 5. A comparison of Poc generation times among the fuzzers.

Target	AFL	AFL++	AFLGo	RegFuzz
cxxfilt	3 h 32 m	3 h 10 m	6 m	6 m
objdump	7 h 44 m	6 h 29 m	35 m	33 m
libming	2 h 31 m	1 h 47 m	1 m	1 m
lrzip	N	11 h 17 m	2 h 10 m	1 h 38 m

5.1.3. Statistical Analysis

In order to prevent accidental factors in the experiment from affecting the performance evaluation of the fuzzers [19], this section shows the statistical p value of the five repeated experimental results of all the evaluations. Where $p1$ represents the p value of RegFuzz to AFL, $p2$ represents the p value of RegFuzz to AFL++, and p3 represents the p value of RegFuzz to AFLGo. As shown in Table 6, in terms of path coverage and code line coverage, almost all $p1$ values and $p3$ values are less than 0.05 and most of them are also less than 0.01. Only $p3$ for libxml2 is slightly higher than 0.05, which indicates that the performance of RegFuzz's path coverage and code coverage is higher than that of AFL and AFLGo, maintaining high stability. $p2$ remains around 0.05 for `cxxfilt, libxml2, libming, and lrzip`, which shows that RegFuzz's program coverage performance is similar to AFL++ for these programs. For `giflib, objdump, and mjs`, $p2$ is greater than 0.05. This is because in the fuzzing experiments of these programs, AFL++ finds more paths and covers more lines of code than RegFuzz. As a directed fuzzer, RegFuzz has low code coverage when the program target location triggers a crash because RegFuzz spends more time in these places. In the programs for which $p2$ is greater than 0.05, RegFuzz triggers many crashes at the program target location; it spends more time on directed fuzzing, resulting in low code coverage, which further causes $p2$ to be greater than 0.05. However, since RegFuzz has found more vulnerabilities, it is acceptable here.

Table 6. The p value in each evaluation.

Targets	Total Paths			Line Coverage		
	$p1$	$p2$	$p3$	$p1$	$p2$	$p3$
cxxfilt	1.5×10^{-4}	6.4×10^{-2}	2.6×10^{-2}	6.5×10^{-4}	7.1×10^{-2}	1.1×10^{-3}
giflib	1.7×10^{-3}	8.0×10^{-1}	3.4×10^{-2}	1.7×10^{-4}	2.8×10^{-1}	4.1×10^{-4}
libxml2	4.2×10^{-5}	2.1×10^{-2}	5.5×10^{-2}	3.3×10^{-6}	4.1×10^{-2}	6.3×10^{-2}
objdump	4.7×10^{-3}	3.9×10^{-1}	2.7×10^{-3}	1.3×10^{-2}	3.6×10^{-1}	3.4×10^{-3}
libming	2.4×10^{-2}	2.3×10^{-2}	5.1×10^{-3}	2.2×10^{-3}	6.1×10^{-2}	4.6×10^{-3}
mjs	8.2×10^{-3}	4.5×10^{-1}	1.2×10^{-2}	4.3×10^{-2}	1.5×10^{-2}	2.7×10^{-2}
lrzip	7.5×10^{-3}	7.0×10^{-2}	6.3×10^{-4}	7.9×10^{-3}	5.7×10^{-2}	7.4×10^{-4}

6. Discussion

We consider RegFuzz to be optimally aligned with those fuzzing scenarios in which users aim to accentuate fuzzing efforts on specific parts of a program while also embracing the possibility of uncovering vulnerabilities in other parts. This efficacy stems from RegFuzz's capacity for directed fuzzing. Furthermore, the tool seamlessly transitions towards fuzzing other parts of the program if the initially designated parts display minimal vulnerabilities or none at all. Such a scenario may transpire when a user initiates the process of familiarizing themselves with a program. Within this context, a user might entertain security concerns regarding a particular component of the program while simultaneously harboring uncertainty about the presence of vulnerabilities in other segments.

While RegFuzz offers the advantages of directed fuzzing to uncover vulnerabilities within specific designated areas, it may not be optimally effective in detecting certain specialized vulnerability types, such as memory leaks or "use after free" (UAF). This limitation arises from the general nature of the seed labels currently employed in RegFuzz, which are not tailored to the characteristics of these specific vulnerability types. In order to address this issue, additional seed labels could be integrated to enhance support for targeting these specialized forms of vulnerabilities, as demonstrated by previous studies [20,21].

Furthermore, both RegFuzz and the prevailing popular fuzzing systems exhibit known limitations. Notably, they are confined to identifying a subset of vulnerabilities and bugs. For instance, their capability to detect issues that do not lead to program crashes, such

as logic implementation errors and user interface (UI) misrepresentations, is notably constrained. This limitation is underscored by a recent investigation [22], which reveals that a considerable portion of common weakness enumeration (CWE) classes remains undiscovered within the context of the OSS-Fuzz project (i.e., they are not identified as fuzzbugs [22]). Moreover, it is noteworthy that merely 9.2% of OSS-Fuzz fuzzbugs align with a corresponding CVE entry for the respective project [22].

7. Related Work

7.1. General Fuzzing

The main objective of general fuzzing is to cover as much program code as possible during fuzzing so as to discover vulnerabilities in each component of the PUT. Usually, researchers enhance fuzzing techniques by focusing on optimizing either the seed mutation strategy or the seed selection strategy. In the context of seed mutation strategy optimization, several approaches have been explored, such as those presented in [4,12,23–29]. These methods aim to identify the most effective mutation techniques and the appropriate positions for seed mutations during the fuzzing process. For instance, MTFuzz [30] utilizes multi-task neural networks to reduce model training costs, and this predicts the coverage of different seeds. This prediction helps guide seed mutation based on the expected coverage. On the other hand, EMS [3] employs a customized probabilistic byte orientation model (PBOM) to probabilistically generate desired mutation byte values based on the input bytes.

Regarding seed selection strategy optimization, there have been significant research efforts in this area as well, as evidenced by the works in [25,31–35]. For instance, MEUZZ [33] employs supervised machine learning to schedule seed selection, predicting which seeds are more likely to yield better fuzzing results. Another notable contribution is K-Scheduler [35], which uncovers the potential edge coverage from seed mutations by examining the CFG and, subsequently, optimizes seed selection scheduling based on these insights.

In contrast to the existing general fuzzing techniques, RegFuzz takes a different approach by dynamically balancing directed fuzzing and general fuzzing. By doing so, it aims to complement general fuzzing techniques and provide a more comprehensive fuzzing solution.

7.2. Directed Fuzzing

Directed fuzzing is an enhanced fuzzing technique designed to focus on fuzzing specific target positions [1,5,8,10,36]. Various approaches have been proposed to improve the efficiency and effectiveness of directed fuzzing. For instance, AFLGo [1] calculates the distance from the seed to the target basic blocks and employs a simulated annealing algorithm to allocate mutation energy to each seed. The seeds closer to the target receive more energy, enabling better exploration of the target area during fuzzing. Savior [8] uses a sanitizer to execute the target and marks potential vulnerability points for monitoring before commencing the fuzzing process on the target location. BEACON [5] utilizes lightweight static analysis calculations to trim redundant paths during runtime before the seed reaches the target, significantly improving the speed of directed fuzzing.

Some directed fuzzers are specifically designed to discover particular types of vulnerabilities, such as UAFuzz [20] and UAFL [21]. These tools are particularly effective in uncovering unique memory vulnerabilities like use after free (UAF) and double free (DF) when targeting such specific issues.

However, when the specified target locations have few vulnerabilities, these directed fuzzers often exhibit low code coverage and a limited vulnerability discovery capability. In contrast, RegFuzz, as proposed here, effectively addresses this limitation by dynamically balancing between directed fuzzing and general fuzzing. By quickly switching to fuzz other areas, RegFuzz ensures that the overall fuzzing process remains comprehensive and thorough, even when encountering regions with limited vulnerabilities or low code coverage.

8. Conclusions

By aiming to fix the shortcomings of the existing directed greybox fuzzing in seed scheduling, we propose a seed prediction method, RegFuzz, which is based on linear regression in directed greybox fuzzing. First, RegFuzz designs several regression labels for seeds from seed code coverage, seed distance, etc. It uses these labels to establish a prediction model to evaluate the seeds. Second, in fuzzing, the seed prediction model is constantly trained to optimize the corresponding label weight values of different seed labels. Finally, the prediction model is used to obtain a score for each seed and optimize the seed scheduling strategy. The experimental results show that RegFuzz has greatly improved code coverage and vulnerability discovery performance when compared to its basis fuzzer AFL, AFL++, which is state-of-the-art fuzzer, and the directed fuzzer AFLGo, which means RegFuzz could gain the merits of both directed fuzzing and ordinary fuzzing.

Author Contributions: Conceptualization, M.L. and Y.Z.; methodology, M.L.; software, M.L.; validation, M.L., Y.Z. and Y.L.; formal analysis, M.L.; investigation, M.L. and Y.L.; resources, Y.Z.; data curation, M.L.; writing—original draft preparation, M.L.; writing—review and editing, M.L., Y.Z. and Y.L.; visualization, M.L.; supervision, Y.Z.; project administration, M.L. and Y.Z.; funding acquisition, Y.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by the Zhejiang Provincial Natural Science Foundation of China under Grant No. LY22F020022, the National Natural Science Foundation of China under Grant No. 61902098, and by the "Pioneer" and "Leading Goose" R&D Program of Zhejiang under Grant No. 2023C03203.

Data Availability Statement: We do not plan to open source the software.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Böhme, M.; Pham, V.T.; Nguyen, M.D.; Roychoudhury, A. Directed greybox fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; pp. 2329–2344.
2. Zalewski, M. American Fuzzy Lop. Available online: https://lcamtuf.coredump.cx/afl/ (accessed on 20 March 2023).
3. Lyu, C.; Ji, S.; Zhang, X.; Liang, H.; Zhao, B.; Lu, K.; Beyah, R. EMS: History-Driven Mutation for Coverage-based Fuzzing. In Proceedings of the 29rd Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 24–28 April 2022; pp. 24–28.
4. Jauernig, P.; Jakobovic, D.; Picek, S.; Stapf, E.; Sadeghi, A.R. DARWIN: Survival of the Fittest Fuzzing Mutators. In Proceedings of the 30th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 27 February–3 March 2023.
5. Huang, H.; Guo, Y.; Shi, Q.; Yao, P.; Wu, R.; Zhang, C. Beacon: Directed grey-box fuzzing with provable path pruning. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, 22–26 May 2022; pp. 36–50.
6. Liang, H.; Zhang, Y.; Yu, Y.; Xie, Z.; Jiang, L. Sequence coverage directed greybox fuzzing. In Proceedings of the 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), Montreal, QC, Canada, 25–26 May 2019; IEEE Computer Society: Washington, DC, USA, 2019; pp. 249–259.
7. Liang, H.; Jiang, L.; Ai, L.; Wei, J. Sequence directed hybrid fuzzing. In Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 18–21 February 2020; pp. 127–137.
8. Chen, Y.; Li, P.; Xu, J.; Guo, S.; Zhou, R.; Zhang, Y.; Wei, T.; Lu, L. Savior: Towards bug-driven hybrid testing. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, 17–21 May 2020; pp. 1580–1596.
9. Yu, L.; Lu, Y.; Shen, Y.; Li, Y.; Pan, Z. Vulnerability-oriented directed fuzzing for binary programs. *Sci. Rep.* **2022**, *12*, 4271. [CrossRef] [PubMed]
10. Chen, H.; Xue, Y.; Li, Y.; Chen, B.; Xie, X.; Wu, X.; Liu, Y. Hawkeye: Towards a desired directed grey-box fuzzer. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 2095–2108.
11. Lee, G.; Shim, W.; Lee, B. Constraint-guided directed greybox fuzzing. In Proceedings of the 30th USENIX Security Symposium, Virtual Event, 11–13 August 2021; pp. 3559–3576.
12. Rawat, S.; Jain, V.; Kumar, A.; Cojocar, L.; Giuffrida, C.; Bos, H. VUzzer: Application-aware Evolutionary Fuzzing. In Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 26 February–1 March 2017; Volume 17, pp. 1–14.
13. Lin, M.; Zeng, Y.; Li, Y. RegFuzz: A Linear Regression-Based Approach for Seed Scheduling in Directed Fuzzing. In Proceedings of the 2023 4th Information Communication Technologies Conference (ICTC), Nanjing, China, 17–19 May 2023; pp. 363–368.

14. Fioraldi, A.; Maier, D.; Eißfeldt, H.; Heuse, M. AFL++: Combining Incremental Steps of Fuzzing Research. In Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT), Boston, MA, USA, 10–11 August 2020.
15. Serebryany, K. Continuous fuzzing with libfuzzer and addresssanitizer. In Proceedings of the 2016 IEEE Cybersecurity Development (SecDev), Boston, MA, USA, 3–4 November 2016; p. 157.
16. Householder, A.D.; Foote, J.M. *Probability-Based Parameter Selection for Black-Box Fuzz Testing*; Technical Report; Carnegie-Mellon University Pittsburgh Pa Software Engineering Institute: Pittsburgh, PA, USA, 2012.
17. Rash, M. afl-cov. 2014. Available online: http://cipherdyne.com/afl-cov/ (accessed on 20 March 2023).
18. Serebryany, K.; Bruening, D.; Potapenko, A.; Vyukov, D. AddressSanitizer: A fast address sanity checker. In Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC), Boston, MA, USA, 12–15 June 2012; pp. 309–318.
19. Klees, G.; Ruef, A.; Cooper, B.; Wei, S.; Hicks, M. Evaluating fuzz testing. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 2123–2138.
20. Nguyen, M.D.; Bardin, S.; Bonichon, R.; Groz, R.; Lemerre, M. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. In Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID), San Sebastian, Spain, 14–15 October 2020; pp. 47–62.
21. Wang, H.; Xie, X.; Li, Y.; Wen, C.; Li, Y.; Liu, Y.; Qin, S.; Chen, H.; Sui, Y. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), Seoul, Republic of Korea, 27 June–19 July 2020; pp. 999–1010.
22. Keller, B.N.; Meyers, B.S.; Meneely, A. What Happens When We Fuzz? Investigating OSS-Fuzz Bug History. In Proceedings of the 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), Los Alamitos, CA, USA, 15–16 May 2023; pp. 207–217. [CrossRef]
23. Lemieux, C.; Sen, K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 475–485.
24. Lyu, C.; Ji, S.; Zhang, C.; Li, Y.; Lee, W.H.; Song, Y.; Beyah, R. MOPT: Optimized mutation scheduling for fuzzers. In Proceedings of the 28th USENIX Security Symposium, Santa Clara, CA, USA, 14–16 August 2019; pp. 1949–1966.
25. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. *IEEE Trans. Softw. Eng.* **2017**, *45*, 489–506. [CrossRef]
26. Chen, P.; Chen, H. Angora: Efficient fuzzing by principled search. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, 21–23 May 2018; pp. 711–725.
27. Yue, T.; Wang, P.; Tang, Y.; Wang, E.; Yu, B.; Lu, K.; Zhou, X. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In Proceedings of the 29th USENIX Security Symposium, Boston, MA, USA, 12–14 August 2020; pp. 2307–2324.
28. Zhu, X.; Feng, X.; Meng, X.; Wen, S.; Camtepe, S.; Xiang, Y.; Ren, K. CSI-Fuzz: Full-speed edge tracing using coverage sensitive instrumentation. *IEEE Trans. Dependable Secur. Comput.* **2020**, *19*, 912–923. [CrossRef]
29. Lin, M.; Zeng, Y.; Wu, T.; Wang, Q.; Fang, L.; Guo, S. GSA-Fuzz: Optimize Seed Mutation with Gravitational Search Algorithm. *Secur. Commun. Netw.* **2022**, *2022*, 1505842. [CrossRef]
30. She, D.; Krishna, R.; Yan, L.; Jana, S.; Ray, B. MTFuzz: Fuzzing with a multi-task neural network. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, 8–13 November 2020; pp. 737–749.
31. Liang, J.; Jiang, Y.; Wang, M.; Jiao, X.; Chen, Y.; Song, H.; Choo, K.K.R. Deepfuzzer: Accelerated deep greybox fuzzing. *IEEE Trans. Dependable Secur. Comput.* **2019**, *18*, 2675–2688. [CrossRef]
32. Li, Y.; Xue, Y.; Chen, H.; Wu, X.; Zhang, C.; Xie, X.; Wang, H.; Liu, Y. Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; pp. 533–544.
33. Chen, Y.; Ahmadi, M.; Wang, B.; Lu, L. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. In Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID), San Sebastian, Spain, 14–15 October 2020; pp. 77–92.
34. Herrera, A.; Gunadi, H.; Magrath, S.; Norrish, M.; Payer, M.; Hosking, A.L. Seed selection for successful fuzzing. In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, 11–17 July 2021; pp. 230–243.
35. She, D.; Shah, A.; Jana, S. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. *arXiv* **2022**, arXiv:2203.12064.
36. Zong, P.; Lv, T.; Wang, D.; Deng, Z.; Liang, R.; Chen, K. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In Proceedings of the 29th USENIX Security Symposium, Boston, MA, USA, 12–14 August 2020; pp. 2255–2269.